

# Assignment #1 - BitList and Book(s)

## Overview

1. **Part 1:** BitList Class
2. **Part 2:** Books

## Preparation

### Create Your Repository

Click on this link: <https://classroom.github.com/a/DdtXzFYv> (<https://classroom.github.com/a/DdtXzFYv>) to accept this assignment in GitHub classroom. This will:

- add you as a member to the course GitHub organization if you are not already a member
- create a repository containing some tests and blank starter files

### Use Git to Clone the Repository

Once you've accepted the assignment:

- you may need to refresh the page before it gives you a link to your repository
- click on that link to see your assignment repo on GitHub

Use the commandline (MacOS has git bundled or WSL for windows) git client or, alternatively, download a git client, such as GitHub Desktop (<https://desktop.github.com/>).

Depending on the client you use (commandline or some other graphical client), the process to clone your repository will vary slightly.

- If you are using the commandline:
  1. create a personal access token (<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>); this will serve as your password
  2. on your repository's page on github, use the green "Code" button on the right side of the screen to copy the Clone with HTTPS to clone (download) the starter files for the homework
  3. use the commandline client to clone your repository (with GITHUB\_REPOSITORY\_URL being the url you copied from the green button): `git clone GITHUB_REPOSITORY_URL`
  4. if prompted for a password, use the personal access token that you created in the first step.
- If you are using GitHub desktop, you can follow the official documentation for cloning a repository (<https://docs.github.com/en/desktop/contributing-and-collaborating-using-github-desktop/adding-and-cloning-repositories/cloning-and-forking-repositories-from-github-desktop>)
  - you may be prompted to sign into github when using this method
  - search for your repository; it should be named `homeworkNN-YOUR_GITHUB_USERNAME`

You can then go through the following steps to commit your first changes. **Note that your repository is private!**

1. once downloaded, go to the project directory, open the `README.md` file... edit it with a text editor of your choice (notepad, TextEdit, Visual Studio Code, Sublime, etc.) so that it includes:
  - your github username
  - the homework number: `Homework #NN`
  - all together: `myusername Homework #01`
2. save your file in your local repository...
  - if using the commandline client
    1. in the same project directory, run `git add README.md` to let git know that you're ready to "save"

2. then save your work locally by running `git commit -m "add homework meta information"` (everything within the quotes after `-m` is any commit message you'd like)
  - regardless of which client you use, please make your commit messages descriptive
  - (what features have been added, what bug has been fixed, etc.)
3. finally, send your work to github... if you're using the commandline client, run `git push`
4. push your work often!
5. ⚠ after you've pushed, go to the repository page on github to **make sure that your latest changes are present**

## Part 1: BitList Class

In a file called `bits.py`, create a class, `BitList`, that represents a series of bits. The class and instances of the class will have methods representing operations that can be performed on those bits.

You can test your class implementation by running the **unit tests** in `bits_test.py`. These unit tests will check that your class and methods behave (*mostly*) as specified by the requirements below.

## Running tests

1. ⚠ Make sure to name your classes and methods exactly as they appear in the instructions below
2. Once you've implemented at least one requirement, try running `test_bits.py` to run automated tests (you can view the code in `test_bits.py` to see what features of your class are being tested
  - note that in some editors, like PyCharm, this may show up as `Run Unittests in...`
3. If your class fulfills the requirements, you should get output similar to this:

```
-----
Ran 27 tests in 0.002s
 
OK
 
Process finished with exit code 0
```

4. If there are errors or test failures, your output will look like this:

```
F.E.....
=====
ERROR: test_arithmetic_shift_right_0 (__main__.TestBitList)
-----
Traceback (most recent call last):
  File "/home/joe/homework/01/test_bits.py", line 33, in test_arithmetic_shift_right_0
    1 + "2"
TypeError: unsupported operand type(s) for +: 'int' and 'str'


=====
FAIL: test_and (__main__.TestBitList)
-----
Traceback (most recent call last):
  File "/home/joe/homework/01/test_bits.py", line 45, in test_and
    self.assertEqual(b1.bitwise_and(b2), BitList('10000001'))
AssertionError: <bits.BitList object at 0x7f7a205c1970> != <bits.BitList object at 0x7f7a205c19d0>

-----
Ran NN tests in 0.002s

FAILED (failures=1, errors=1)
```

- an `F` in the first line represents a failure (the test expected a certain result, but the method, function, or other implemented code gave back a different result)
  - an `E` means there was an exception - a runtime error
  - an `.` means that the test passed
  - below the first line, each error or failure is detailed
5. of course, your goal is to get zero errors or failures
- just because your class passes all of the tests, it doesn't mean it's working perfectly, as the tests are not comprehensive
  - it's best to test often and build incrementally so that you don't end up with a large implementation that has fundamental flaws that may have been caught by earlier testing

## Define custom errors; causing and handling runtime errors (Exceptions)

1. to cause a runtime error in your program, use the keyword, `raise` followed by an instance of an exception:
  - for example, `TypeError` is the name of a built-in Exception
  - to *cause* a `TypeError`, use `raise` followed by an instance... with the instance created by passing in a string describing the error
  - `raise TypeError('type BitList does not support the + operator')`
2. instead of using built in exceptions, you can create your own exceptions!
  - see the python docs (<https://docs.python.org/3/tutorial/errors.html#user-defined-exceptions>)
  - essentially, inherit from the base exception class using this syntax: `class MyException(Exception)`
  - the body of the class can simply be `pass ...` from the docs (<https://docs.python.org/3/tutorial/controlflow.html#highlight=pass#pass-statements>), `pass` simply does nothing but fulfills syntax requirements (in particular, for an indented block of code where you want nothing to happen)
  - for example: `class FooError(Exception): pass`
3. Why use custom exceptions?
  - it allows users of your class / method / function to gracefully recover from an error by using `except` with a specific type of Exception
4.  **Create two exception classes** at the top of your `bits.py` file (both should inherit from `Exception` and have `pass` as the body:
  1. `DecodeError` - raised when there's an issue attempting to decode a series of bits as a particular encoding
  2. `ChunkError` - raised when a series of bits can be split up into evenly sized chunks of bits

## Create the `BitList` class; it should support the following behavior:

### `BitList(s)`

Create a new series of bits from a string that represents a binary number:

- the string should only consist of 0's and 1's
- if it does not, `raise` a `ValueError` with a message stating that only 0 and 1 are allowed in the string
- if the string entered is valid, find some way to retain the bits
  - save the data entered on `self` (that is, keep an internal representation of the data for each instance)
  - choose a data type for the value that will best serve you

Example Usage:

```
b = BitList('10000011')
```

```
try:
    b = BitList('FE02')
except ValueError:
    print('Format is invalid; does not consist of only 0 and 1')
# Format is invalid; does not consist of only 0 and 1
```

==

If both `BitList` instances contain the same series of bits, then they're equal:

```
b1 = BitList('11000001')
b2 = BitList.from_ints(1, 1, 0, 0, 0, 0, 0, 1)
print(b1 == b2) # True!
```

To use this operator, define a method:

```
def __eq__(self, other):
    return some_boolean
```

If `__eq__` is implemented, then the `==` operator can be used to test for equality on instances of this class. Treat `other` as the other operand of `==` (so, it would be the *other* `BitList` being compared to *this* instance, almost as if: `my_instance == other`).

### `BitList.from_ints(b1, b2, ...bN)`

A new series of bits can also be created by supplying integers directly to a `@staticmethod` `from_ints`:

- raise a value error there are digits other than 0 and 1
- notice that `from_ints` is called from the class itself (rather than an instance); it's a static method, so decorate with `@staticmethod` by adding `@staticmethod` above the method definition

```
def m(self):
    pass
```

- calling `from_ints` returns a new `BitList`.

As shown below, adding `@staticmethod` allows `from_ints` to be called on the name of the class itself rather than just on an instance! This is similar to static methods in Java.

```
b = BitList.from_ints(1, 1, 0, 0, 0, 0, 0, 1)
print(type(b)) # <class 'bits.BitList'>
print(b) # 11000001
```

```
try:
    b = BitList.from_ints(1, 2, 3, 4)
except ValueError:
    print('Format is invalid; does not consist of only 0 and 1')
```

### Converting to a `str`

Implement the appropriate method on the `BitList` class such that converting to a list (such as when using `str` or printing out a value) a `BitList` instance displays every bit in the series of bits:

- there are no spaces between each bit

```
b = BitList('10000011')

str(b)    # 10000011
print(b)  # outputs 10000011
```

### `.arithmetic_shift_left()` and `.arithmetic_shift_right()`

This method **has no parameters** and **it does not return a value**.

The bits in the series can be shifted to the left or right by one.

1. for shift left
  - the left most bit should be discarded
  - a zero should be added to the right
  - $0110 \rightarrow 1100$
2. for shift right
  - the right most bit should be discarded
  - the left most bit should be duplicated
  - $1100 \rightarrow 1110$

Note that the internal representation of bits should change. That is, there is no return value; instead, the actual `BitList` instance changes:

```
b = BitList('10000010')
b.arithmetic_shift_right()
print(b) # 11000001
```

### **.bitwise\_and(otherBitList)**

This method has **one parameter, another instance of `BitList`**, and it returns a new `BitList` instance.

A bitwise and can be performed with an *incoming* `BitList` :

- a bitwise and can only be performed if both sequences of bits are of equal length
- for every position, use a logical `and` to produce a new bit
  - treat 1 as `True`
  - treat 0 as `False`
  - use these boolean values with a logical `and` evaluate to either `True` or `False`
  - consequently, performing a bitwise and on `1110` and `1011` produces `1010`

```
1110 (TTTF) - Operand 1
1011 (TFTT) - Operand 2
1010 (TFTF) - Result
```

- optionally, simply multiplying each position also works!
- again, the method itself has one parameter, the other `BitList` instance
- it returns a new `BitList` instance

```
b1 = BitList('10000011')
b2 = BitList('11000001')
b3 = b1.bitwise_and(b2)
print(b3) #10000001
```

### **.chunk(chunk\_length)**

This method has a **single parameter**, an `int` representing the length of the chunks that the instance of `BitList` should be split up by.... and it **returns a list of lists, with each sub list containing bits**:

```
b = BitList('01000011')
print(b.chunk(4))
# the BitList is broken up into chunks of 4 bits
# ... resulting in a list of lists, with each sub list containing 4 bits:
# [[0, 1, 0, 0], [0, 0, 1, 1]]
```

If there's an error, this method should cause a runtime exception, a `ChunkError` ... instantiate the error with a message describing the cause of the error

```
b = BitList('0101010101')
print(b.chunk(4)) # <--- raises a ChunkError!
```

## **.decode(encoding='utf-8')**

This method has a **single parameter**, the encoding (a `str`), and it **returns a string**:

1. the encoding can only be `us-ascii` or `utf-8`
  - the default encoding should be `utf-8`
  - consequently, `decode` can be called with no arguments
2. it returns a string

An instance of `BitList` can be decoded. In the example below, the bits are treated as 7-bit ASCII (`us-ascii`):

```
bits = BitList.from_ints(1, 1, 0, 0, 0, 0, 1)
ch = bits.decode('us-ascii')
print(ch) # a
```

This should work if there are multiple characters encoded:

```
new_bit_list = BitList('11000011000001')
s = new_bit_list.decode('us-ascii')
print(s) # aA
```

One way to implement this is to:

1. calculate the decimal value from the bits
2. convert the decimal value into a character with `chr`

This is a little tricky, as chunking bits into equivalent lengths isn't adequate:

- `utf-8` is variable length
- the "leading byte" must be examined to determine how many bytes total are required
- for example, encountering the byte `11101010` means that there should be three bytes total (there are 3 1's)
- the continuation bits are all prefixed with `10`
- see the slides / notebook on `unicode` for more details

```
b = BitList('11110000100111111001100010000010111000101000001010101100')
s = b.decode('utf-8')
print(s) # 🤪€
```

To deal with decoding errors:

- if the encoding passed into `decode` is not `us-ascii` or `utf-8`, immediately raise a `ValueError`, instantiated with a message stating that the encoding is not supported
- if the leading byte is invalid (see wikipedia's table (<https://en.wikipedia.org/wiki/UTF-8#Encoding>) for valid bit patterns to start a leading byte), cause a runtime error by throwing an `DecodeError` exception (the value passed in to the construction of this error should describe that the error was due to an invalid leading byte)
- if a continuation byte is invalid (it doesn't start with `10`, for example) cause a runtime error by throwing an `DecodeError` exception (the value passed in to the construction of this error should describe that the error was due to an invalid continuation byte)

```
# invalid continuation byte

b = BitList('1111000000011111001100010000010')
try:
    print(b.decode('utf-8'))
except DecodeError:
    print('error!')
```

```
# invalid leading byte

b = BitList('10000011')
try:
    print(b.decode('utf-8'))
except DecodeError:
    print('error!')
```

## Part 2: Books

Using at least two books from project Gutenberg (<https://www.gutenberg.org/>) use file io and basic Python to manipulate the content of the books.

## Gather Your Data

- "manually" download `txt` versions of at least two books
- (right-click save as on the link – you do not have to have to programmatically download these books)
- save the books into the `data` folder of your repository

## Working with Files

1. start jupyter lab on the commandline in the root of your cloned project directory directory: `jupyter-lab`
2. open jupyter lab in your browser; use the file explorer to find, open, and edit the notebook `book.ipynb`
3. in a markdown cell:
  - write out information about the books you downloaded
    1. a description of the books (titles and authors, for example... or all works by author, etc.)
    2. the url to the book or folder of books
    3. any license information
    4. encoding if provided
  - write a question you'd like answered about the books that might require some programming... for example:
    1. who was the most frequently mentioned character in *Pride and Prejudice*... and in *Sense and Sensibility*?
    2. are the pronouns in Mary Shelley's books predominantly male or female?
    3. etc. .... (feel free to use either of the above questions, or come up with your own!)
4. open the `txt` versions of the books that you downloaded by writing Python in your notebook
  - use a relative path to `data/name_of_your_file` (this should work on both MacOS and Windows despite the path separator differences)
  - alternatively, you can use `os.path.join` (<https://docs.python.org/3/library/os.path.html#os.path.join>) to construct paths with the right path separator
5. use jupyter notebook code cells to analyze the text in your file...and come up with an answer
  - **a dictionary** (or counter object if you're familiar with counters) must be part of your analysis
  - note that dictionaries can be used as counters by having the key be the item being counted... and the value being the count
    - checking for an existence of a key with `in` may be helpful
    - "catching" a `KeyError` will also identify keys that are missing
    - adding a key looks like: `d['new key'] = 'new value'`

- update a key looks like: `d['key'] = d['key'] + 1`
  - (or again, a counter object can be used)
6. document your steps in markdown cells
  7. your answer does not have to be correct (or even exact!)... as long as you have some description of your analysis
  8. again, make sure you include the txt files in your repository within the `data` directory